湖南大學
HUNAN UNIVERSITY

课程实验报告

课　程 名 称：编译原理

实验项目名称：词法分析程序

专　业 班 级：计科 1706 班

姓　　名：李立天

学　　号：201708010806

指 导 教　师：全哲

完 成 时 间：2019 年 10 月 20 日

信息科学与工程学院

一、实验目的

学习和掌握词法分析程序手工构造状态图及其代码实现方法。

二、实验任务

（1）阅读已有编译器的经典词法分析源程序；
（2）用 C 或 C++语言编写一门语言的词法分析器。

三、实验内容

（1）阅读已有编译器的经典词法分析源程序。

选择一个编译器，如：TINY，其它编译器也可（需自备源代码）。阅读词法分析源程序，理解词法分析程序的手工构造方法——状态图代码化。尤其要求对相关函数与重要变量的作用与功能进行稍微详细的描述。若能加上学习心得则更好。TINY 语言请参考《编译原理及实践》第 2.5 节（见压缩包里附带的文档）。

（2）确定今后其他实验中要设计编译器的语言，如 TINY 语言，又如更复杂的 C－语言（其定义在《编译原理及实践》附录 A 中）。也可选择其它语言，不过要有该语言的详细定义（可仿照 C－语言）。一旦选定，不能更改，因为要在以后继续实现编译器的其它部分。鼓励自己定义一门语言。

（3）根据该语言的关键词和识别的词法单元以及注释等，确定关键字表，画出所有词法单元和注释对应的 DFA 图。

（4）仿照前面学习的词法分析器，编写选定语言的词法分析器。

（5）准备 2~3 个测试用例，要求包含正例和反例，测试编译结果。

四、实验步骤

1. 确定在今后的实验中实现一个 C-语言编译器

选择 Cminus 语言的原因有 3 点：

① 我对 C 语言较为熟悉，且 C 语言是一门经典的语言；
② C 子集语言具有完整的词法、语法、语义定义，便于完成实验；
③ C 子集较 TINY 语言更有难度，可以挑战自己的工程能力。

2. 学习 TINY 语言的词法分析源程序

(1) 了解编译器的文件结构

通过仔细阅读 TINY 编译器的源代码，发现词法分析部分主要由 main.c, scan.c,, globals.h, util.c 等四个文件组成，它们的主要功能如下：

main.c – 处理命令行参数，调用词法分析模块
globals.h – 定义了词法单元的 enum 类型和其它全局变量
scan.c – 实现了词法分析功能
util.c – 相关工具，包含打印词法单元信息

(2) 理解相关函数和变量的作用

scan.c

```
StateType state;                    //当前 DFA 的状态
```

```
TokenType currentToken;    //从当前状态推断的词法单元的类型
char lineBuf[BUFLEN];      //从文件中读入一行作为缓冲区
TokenType getToken(void);       //读入下一个词法单元，并返回词法单元的类型
TokenType reservedLookup (char * s)      //若词法单元为 ID，需要进一步判断它是不是关键字
int getNextChar(void);       //从缓冲区读取一个非空白字符
void ungetNextChar(void);  //DFA 输入错误时，用于回退一个字符
```

Globals.h

```
extern FILE* source;              //源代码文件
extern FILE* listing;             //输出 tsxt 文件
extern int lineno;                //当前行号
typedef enum {}TokenType;       //枚举类型的保留字
```

util.h

```
printToken(TokenType token, const char* tokenString)          //输出所识别的词法单元
```

(3) 学习词法分析器的构造基本步骤

A. 通过语言的完整定义，总结出语言的词法单元集合。如图所示，TINY 语言具有 4 大类型的记号：保留字 reserved word、特殊符号 opeartion、数字类型 NUM、标识符类型 ID。

表2-1　TINY语言的记号

保 留 字	特 殊 符 号	其　他
if	+	**数**
then	-	(1个或更多的数字)
else	*	
end	/	
repeat	=	
until	<	**标识符**
read	(	(1个或更多的字母)
write	)	
	;	
	:=	

B. 根据语言的记号和相应的正则表达式，构建 DFA 状态图。其中有几个注意事项：

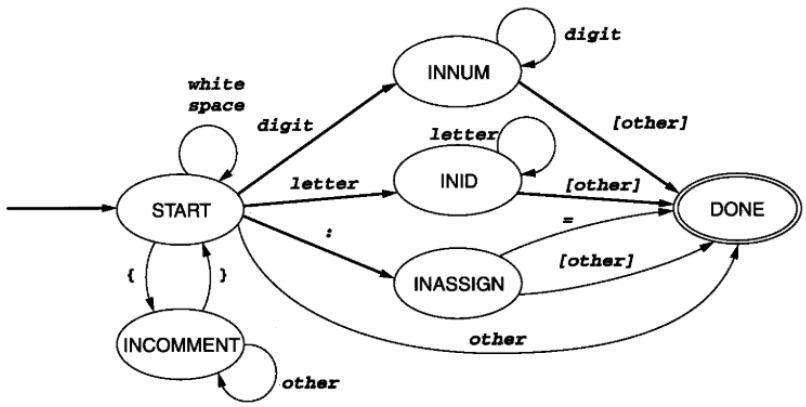


图2-8　TINY扫描程序的DFA

①可以分别为不同种类的符号各自画出 DFA 图，最后合成一个 DFA 图；

②对返回值的处理：根据字符串在 DFA 中的路径，可以记录它当前所处的词法单元类型

currentTokenType，到达 DONE 状态时，返回此词法单元的类型；
③对注释的处理：{}为注释符号，不能存在嵌套，且该程序不保存注释；
④对空白符的处理：制表符、空格、回车被当做空白符处理。处理的方法是，在 START 状态输入一个空白符时，DFA 仍然停留在 START 状态。
⑤对保留字的处理：不单独为保留字设置 DFA 状态图，创建枚举类型来保留关键字；读取由字符构成的 ID，该 ID 识别结束后，在枚举类型中查找是否是保留字，如果是保留字，则做特殊处理。
⑥两种获取下一个字符的方式：第一种是直接消耗掉输入符号，如果在识别一个记号的过程中，能够确定输入的字符属于该记号，则直接消耗掉输入符号；第二种是不消耗输入符号，如果在识别一个记号的过程中，读到某一个输入符号能确定该记号读取完毕，但是该符号并不属于该记号时，该符号不能被消耗。Tiny 在区别这两种方式的方法是在 DFA 中添加[other]边，如果是通过[other]边到达接受符号，那么表示该[other]符号需要被回吐。

C. 使用手工构造的方法将 DFA 代码化，伪代码如下：

```
TokenType getToken()              //每次返回一个词法单元
{
    currentState = START;         //初始状态
    bool save;
    while(state != DONE)          //直到到达 DONE 状态
    {
        c = getNextChar();        //输入一个字符
        save = true;

        /*  使用 switch 语句，根据字符 c 更新以下变量  */
        switch(c):
        currentState = nextState(c); //当前状态
        currentToken = nextToken(c);    //当前词法单元的类型
        save = isSave();                //判断是否需要保留此字符
    }

    if(save) tokenString += c;          //如果保存，则让记号加上该字符
    printToken();                       //打印词法单元
    return currentToken;
}
```

3. 为 Cminus 语言的代码化工作做准备

(1) 根据语言的定义，得到一张词法单元表：

A.1 C－惯用的词法

1. 下面是语言的关键字：

```
else if int return void while
```

所有的关键字都是保留字，并且必须是小写。

2. 下面是专用符号：

```
+ - * / < <= > >= == != = ; , ( ) [ ] { } /* */
```

3. 其他标记是***ID***和***NUM***，通过下列正则表达式定义：

```
ID = letter letter*
NUM = digit digit*
letter = a|..|z|A|..|Z
digit = 0|..|9
```

小写和大写字母是有区别的。

4. 空格由空白、换行符和制表符组成。空格通常被忽略，除了它必须分开 ***ID***、***NUM***关键字。
5. 注释用通常的C语言符号**/*...*/**围起来。注释可以放在任何空白出现的位置（即注释不能放在标记内）上，且可以超过一行。注释不能嵌套。

词法单元的分类	TokenType	tokenString
关键字	ELSE	else
	IF	if
	INT	int
	RETURN	return
	VOID	void
	WHILE	while
专有符号	PULS	+
	MINUS	-
	TIMES	*
	OVER	/
	LT	<
	LE	<=
	GT	>
	GE	>=
	EQ	==
	NEQ	!=
	ASSIGN	=
	SEMI	;
	COMMA	,
	LPAREN	(
	RPAREN	)
	LBRACKET	[
	RBRACKET	]
	LBRACE	{
	RBRACE	}
数字	NUM	
标识符	ID	

(2) 根据各个语法单元的正则定义，得出 DFA：

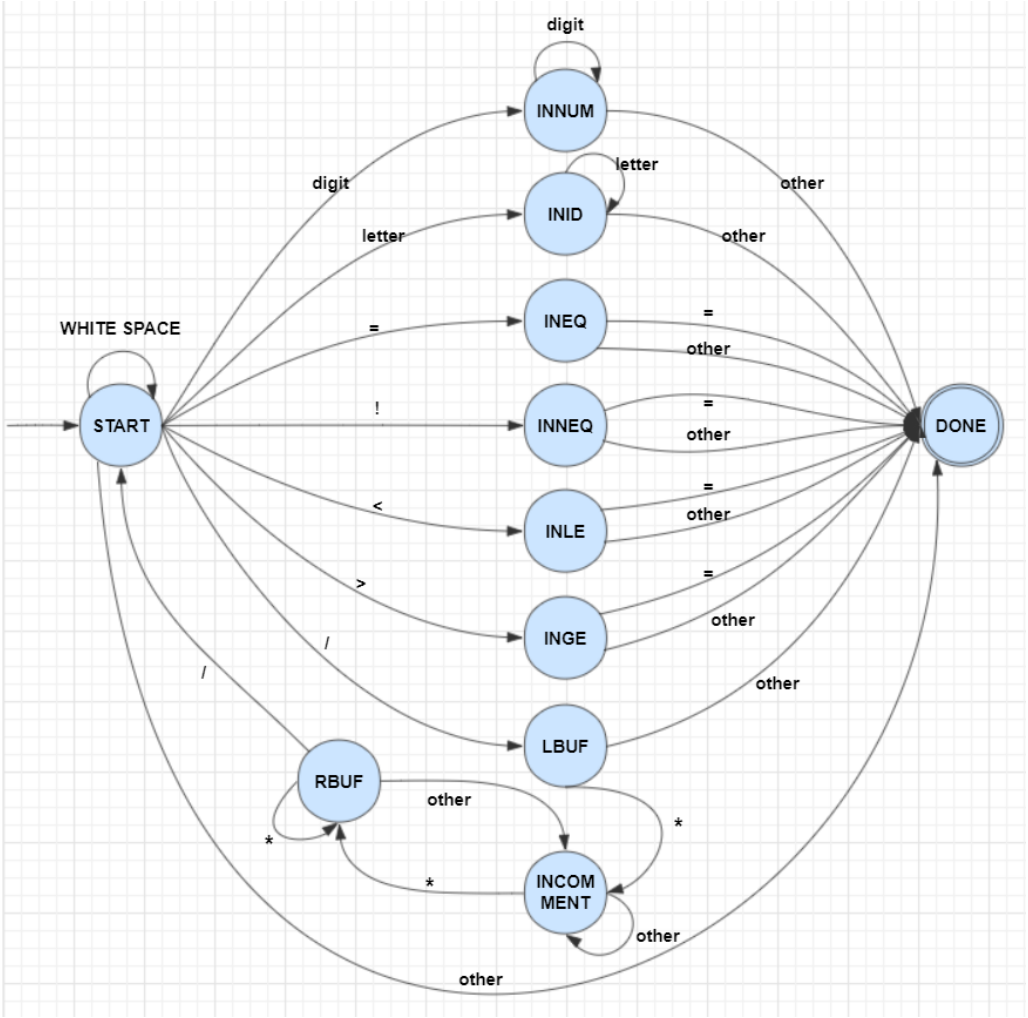
A. 各个词法单元的正则定义如下：

- ID 由字母组成，区分大小写；
- NUM 由若干个数字组成；
- 专用符号采取最长匹配原则，例如“<”和“<=”识别为“<=”；
- 注释由/*、*/及它们之间的符号构成，不允许嵌套

B.对于构建 DFA 进行的一些规定：

对于由一个符号构成的专用符号：+ - * ; , ( ) [ ] { }合并为 others 边，在 START 状态中输入以上符号，直接转移到 DONE 状态；
对于由两个符号构成的专用符号：<= 、>= 、==、!=，需要一个中转状态来表示已经输入了第一个字符。如果输入的第二个字符属于该记号，则转到 DONE 状态，同时记录该双字符符号；如果不属于该记号，则需要回退一个字符，到达 DONE 状态，记录单字符符号；
对于“/”字符：输入该字符后转到 LBUF 状态，等待后续输入来判断它是注释还是除法符号；
对于输入的不属于记号的字符：使用[others]来表示，告诉词法分析器这里需要回退一个字符。
对于注释：输入/和*之后将进入到 INCOMMENT 状态，此状态有 others 自旋边，如果再输入一个*，将到达 RBUF 状态，等待一个“/”结束注释。注释结束之后将回到 START 状态，词法分析器并不负责记录注释的字符串；

C. 最后构建出来的总 DFA 如图所示：



4. 将 DFA 代码化得到 getToken()函数：

```
/*  创建词法单元的类型  */
enum TokenType
{
    ENDFILE, ERROR,
    /*  保留字  */
```

```
    ELSE, IF, INT, RETURN, VOID, WHILE,
    /* 多字符词法单元 */
    ID, NUM,
    /* 特殊字符 */
    PLUS, MINUS, TIMES, OVER, LT, LE, GT, GE, EQ, NEQ, ASSIGN, SEMI, COMMA,
LPAREN,     RPAREN, LBRACKET, RBRACKET, LBRACE, RBRACE
};

/* 创建 DFA 状态的类型 */
enum StateType
{ START, LBUF, RBUF, INCOMMENT, INNUM, INID, INEQ, INLE, INGE, INNEQ, DONE };
```

GetToken 函数的伪代码在上面已经展示过了，Cminus 词法分析器采用了与 TINY 相同的思想。在这里我使用了 C++代替 C 语言，方便易用的同时，也优化了部分实现方式。

5. 其它关键函数的实现

A. 判断 ID 是否为关键字，并返回其正确类型：使用 unordered_map 哈希表，替换原来的线性搜索算法，将时间复杂度由 O(N)降低至 O(1).

```
/* 关键字表 <关键字，类型> */
unordered_map<string, TokenType> reservedMap =
{
    {"else", ELSE}, {"if", IF}, {"int", INT},
    {"return", RETURN}, {"void", VOID}, {"while", WHILE}
};

/* 查关键字表，判断字符串的 TokenType 是 ID 还是 reserved word */
TokenType reservedLookup(string s)   {
    return (reservedMap.find(s) == reservedMap.end()) ? ID : reservedMap[s];
}
```

B. 缓冲区的读入、回退的实现：由于使用了 getline()函数，对一行中的最大字符数没有限制。

```
/* 从缓冲区获取下一个字符 */
char getNextChar(void)
{
    /* 当前行读取完毕，尝试读取下一行 */
    if (linepos >= lineBuf.length())
    {
        lineno++;
        if (getline(source, lineBuf))
        {
            lineBuf += "\n";
            if (EchoSource) cout << setw(4) << lineno << ": " << lineBuf;
            linepos = 0;
            return lineBuf[linepos++];
```

```
            }
            else
            {
                EOF_flag = true;
                return EOF;
            }
        }
        else return lineBuf[linepos++];
}

/*  回退一个字符  */
void ungetNextChar()
{
        if (!EOF_flag) linepos--;
}
```

6. 测试数据及测试结果

测试样例 1：1.cminus

样例说明：欧几里得算法的 Cminus 语言实现；

```
/* A program to perform Euclid's Algorithm to compute gcd. */
int gcd(int u, int v)
{
        if (v == 0) return u;
        else return gcd(v, u - u / v * v); /* u-u/v*v == u mod v */
}

void main(void)
{
        int x;
        int y;
        x = input();
        y = input();
        output(gcd(x, y));
}
```

测试结果：如下图所示，通过命令行调用 CminusLex 对 1.cminus 源程序进行了语法分析。

对于源程序的第一行：由于是注释，因此自动被语法分析器忽略；

对于源程序的第二行：语法分析器能够正确地识别出：保留字 int，标识符 gcd，特殊符号“（”，标识符 u、v，特殊符号“）”等；

对于其他行：分析结果符合实际情况，验证成功。

```
选择C:\Windows\System32\cmd.exe
Microsoft Windows [版本 10.0.17763.805]
(c) 2018 Microsoft Corporation。保留所有权利。

D:\大三上\编译原理\实验\实验二\CminusLex\Debug>CminusLex 1.cminus
   1: /* A program to perform Euclid's Algorithm to compute gcd. */
   2: int gcd(int u, int v)
        2: reserved word: int
        2: ID, name = gcd
        2: (
        2: reserved word: int
        2: ID, name = u
        2: ,
        2: reserved word: int
        2: ID, name = v
        2: )
   3: {
        3: {
   4:   if (v == 0) return u;
        4: reserved word: if
        4: (
        4: ID, name = v
        4: ==
        4: NUM, val = 0
        4: )
        4: reserved word: return
        4: ID, name = u
        4: ;
```

测试样例 2：2.cminus

样例说明：源程序中列出了所有可识别的保留字、特殊记号，并对 NUM 和 ID 进行了举例；

```
/* reserved words */
else if int
return void while

/* operation */
+ - * / < <= > >=
== != = ; , ()
[] {} /* */

/* ID and NUM */
number 806
string JXX
```

实验结果：

如右图所示，程序能够识别出所有的 reserved words 和 special operations。并且识别出了 NUM = 806，ID = “JXX”；在源程序的文件尾，分析出了结束符 EOF。

```
D:\大三上\编译原理\实验\实验二\CminusLex\Debug>CminusLex 2.cminus
   1: /* reserved words */
   2: else if int
        2: reserved word: else
        2: reserved word: if
        2: reserved word: int
   3: return void while
        3: reserved word: return
        3: reserved word: void
        3: reserved word: while
   4:
   5: /* operation */
   6: + - * / < <= > >=
        6: +
        6: -
        6: *
        6: /
        6: <
        6: <=
        6: >
        6: >=
   7: == != = ; , ()
        7: ==
        7: !=
        7: =
        7: ;
        7: ,
        7: (
        7: )
   8: [] {} /* */
        8: [
        8: ]
        8: {
        8: }
   9:
  10: /* ID and NUM */
  11: number 806
        11: ID, name = number
        11: NUM, val = 806
  12: string JXX
        12: ID, name = string
        12: ID, name = JXX
        13: EOF
```

测试样例 3：3.cminus
样例说明：源程序中故意设置了几处错误语法，例如不存在的字符。

```
/* bad examples */
void string2int()
{
    a += b;
    a = ~(!b);
    b := a ^ b;
}
```

实验结果：
预期的函数名 string2int 被解析为 ID“string”，NUM“2”以及 reserved word“int”。因为语法分析器不允许 ID 中出现任何非字母的字符；
符号“+ =”被解析为两个独立的运算符，原因是在 Cminus 中只有“+”和“=”，而没有“+ =”；
符号“: =”被解析为 Error 和“=”，原因是“：”在 Cminus 中没有定义；
符号“!”，“~”，“^”等视为报错，原因是 Cminus 中没有定义取反，异或等符号。
综上，语法分析器能够识别出不合语法规范的句子，并能够提示错误的字符信息。

```
D:\大三上\编译原理\实验\实验二\CminusLex\Debug>CminusLex 3.cminus
   1: /* bad examples */
   2: void string2int()
        2: reserved word: void
        2: ID, name = string
        2: NUM, val = 2
        2: reserved word: int
        2: (
        2: )
   3: {
        3: {
   4:   a += b;
        4: ID, name = a
        4: +
        4: =
        4: ID, name = b
        4: ;
   5:   a = ~(!b);
        5: ID, name = a
        5: =
        5: ERROR: ~
        5: (
        5: ERROR: !
        5: ID, name = b
        5: )
        5: ;
   6:   b := a ^ b;
        6: ID, name = b
        6: ERROR: :
        6: =
        6: ID, name = a
        6: ERROR: ^
        6: ID, name = b
        6: ;
   7: }
        7: }
        8: EOF
```

五、实验心得

1) 此次实验增强了我对词法分析器过程的了解同时也引发了我对词法分析器以及语言定义的一些思考。
2) 本次实验分为两个部分，第一部分是对 Tiny 的词法分析器中源代码进行学习；第二部分是自己构建 C-语言的词法分析。如果没有第一部分中对 Tiny 语法分析器的学习，在第二部分是很难自己构建出 C-语言的词法分析器的。原本是想自己来尝试以自己的思路去编写，但是读了 Tiny 的源代码之后发现其使用的方法肯定会比我自己编写更高效。
3) Tiny 已经是一个完整的编译器了，其中构架是比较完善的，担心自己写了词法分析器后，与后面的实验可能会承接不上，所以选择以 Tiny 的架构来编写自己的词法分析器。除了学习其中编译器的架构之外，还学会了通过状态图去构建词法分析器。相比想象中读取一个字符然后判断这个字符和前面的其他字符可能构建出来哪些词法单元这种方法来说，用 DFA 构建词法分析器无疑是一个更好的方法。DFA 向我们更清晰地展示了词法单元的读取过程，在查找词法分析器中的错误也可以更加清晰明了。
4) 词法分析器的精华部分在于状态转换图，如果能够清晰的画出状态转换图，词法分析器的构造就完成了一大半。但是状态转换图的构造并不是十分容易，在此次状态转换图构造过程较为简单，这是因为此次使用的 C-语言，其已经将 C 语言简化了很多。如果构建更加复杂的语言的状态转换图，难度还是较大的。与此同时此次的 C-语言的状态转换图也足以以小见大。例如在>=的分析过程中，如果没有前人已经规定好的最长子序列优先原则，那么我可能就会去纠结应该将其分析为大于等于还是大于号和等于号这个问题了。换句话来说，定义一个语言要考虑的问题也很多，其是否会有歧义？是否不能构造出一个正确的 DFA？等等。
5) 在代码实现 DFA 的过程中，同样遇到了一些问题，例如，/与/*的分析，前者是除号会被保存在词法单元，后者是注释的开始符号将被词法分析器忽视。按照最长子序列优先原则，可以解决分析为哪个符号。但是问题出现在输入符号的保存过程中。因为读取到/后面的一个符号才能够直到当前符号串为/还是/*，在读入/符号时就不能确定/符号是否应该保存。最终想到的办法是在第二个符号确定后，如果不是*那么就将/另外加入词法单元。这样虽然解决了问题，却破坏了原来程序的简洁明了的结构。